Parallel Programming Lab

01 - Introduction to CUDA

Filippo Ziche (filippo.ziche@univr.it)

University of Verona - Department of Engineering for Innovation Medicine
PARCO - Parallel Computing Lab

March 4, 2025

Agenda

- What is CUDA?
- Calling a Device Functions
- CUDA Concepts and Keywords
- Memory Management
- Error handling
- CUDA Compiler
- Exercises!

What is CUDA?

- CUDA (**C**ompute **U**nified **D**evice **A**rchitecture) is a parallel computing platform and programming model that allows to use a GPU for general purpose computing.
- It is small extension of C/C++ language.
- It allows you to accelerate your C/C++ code by moving the computationally intensive portions of your code to an NVIDIA GPU.

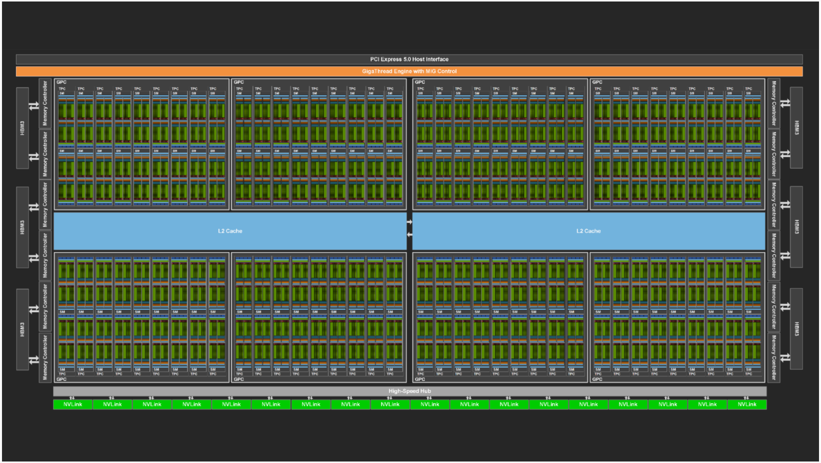
Calling a Device Function (Kernel)

- A kernel is a function callable from the host and executed on the CUDA device. They are the heart of your CUDA code.

```
__global__ void my_kernel(int* x, int N) { ... }
int main() {
    ...
    my_kernel<<<blocks, threads>>>(x, N);
    ...
}
```

- A kernel is defined using the __global__ declaration specifier. They must have return type void.
- The number of CUDA threads that execute a kernel is specified using a new <<< ... >>> execution configuration syntax

CUDA Architecture Overview



A lot of concurrent threads.

Built-in Variables

CUDA provides a set of built-in variables to facilitate the mapping between threads and data. In the case of 1-dimensional data we have the following:

- **threadIdx.x** Thread index within the block
- **blockIdx.x** Block index within the grid
- **blockDim.x** Dimension of the block (number of threads)
- **gridDim.x** Dimension of the grid (number of blocks)

Common pattern to get global id of the thread:

```
gid = blockIdx.x * blockDim.x + threadIdx.x;
```

Common CUDA Code Organization

1. Initialize host data structures
2. Allocate device data structures (cudaMalloc)
3. Copy host data structures to device (cudaMemcpy)
4. Invoke kernel function
5. Copy device result to host (cudaMemcpy)
6. Free host and device data structures (freeCuda)

CUDA Memory Management I

- Allocate memory on the device.

  `__host__ __device__ cudaError_t cudaMalloc(void** devPtr, size_t size )`

- Copies data between host and device.

  `__host__ cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)`

- Frees memory on the device.

  `__host__ __device__ cudaError_t cudaFree(void* devPtr)`

- Allocates memory that will be automatically managed by the Unified Memory system.

  `__host__ cudaError_t cudaMallocManaged(void** devPtr, size_t size , unsigned int flags )`

CUDA Memory Management II

```
...

int* devArray;
int size = 120;
cudaMalloc(&devArray, size * sizeof(int));

// Send data to GPU
cudaMemcpy(devArray, hostArrayA, size * sizeof(int),
    cudaMemcpyHostToDevice);

// Process data
kernel<<<...>>>(devArray, size, ...);

// Get result
cudaMemcpy(hostArrayB, devArray, size * sizeof(int),
    cudaMemcpyDeviceToHost);

cudaFree(devArray);

...
```

CUDA Memory Management III

```
...

int* unifiedArray;
int size = 120;

// Allocate on unified memory
cudaMallocManaged(&unifiedArray, size * sizeof(int));

// Process data
kernel<<<...>>>(unifiedArray, size, ...);

// Do somethind with the processed data
host_func(unifiedArray, size);

cudaFree(devArray);

...
```

CUDA Error Handling

- Every CUDA call (except kernel launches) return an error code of type cudaError_t

```
cudaError_t err = cudaMalloc(&fooPtr, -1);
if (cudaSuccess != err)
    printf("Error : %d\n", err);
```

- CUDA kernel invocations do not return any value. Error from a CUDA kernel call can be checked after its execution by calling cudaGetLastError()

```
fooKernel<<<b,t>>>();
cudaDeviceSynchonize(); // Remember to sync!
cudaError_t err = cudaGetLastError();
...
```

CUDA Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver, It automatically invokes all the necessary tools and compilers like cudacc and g++
- Any executable with CUDA code requires the CUDA runtime library (cudart)
- Simple usage of NVCC to compile:

```
nvcc -std=c++11 <source> -o <executable>
```

Exercises

Clone the repository at
`https://github.com/z1ko/parallel_prog_course`

Today you have the following exercises:

1. **hello_world**: Simple program to test compilation.
2. **vector_set**: Initialize a vector with a value.
3. **vector_add**: Add two vectors.